Cairo University
Faculty of Engineering
Computer Eng. Dept.
CMPS103

Data Structures and Algorithms
LAB #4
Recursion & Binary Search Trees

Objectives

By the end of this lab, the student should be able to:
- Explain how recursion occurs in memory.
- Extract basic components of a recursive function from the problem description.
- Implement and trace recursive functions.
- Explain the tree data structure, storage and usage.
- Know the difference between tree traverse methods.
- Write algorithms for binary search tree applications.

Part I – Recursion

What is recursion?
A function is said to be recursive if it calls itself, below is implementation for a recursive function that prints the phrase “Hello World” a total of *count* times:

Example 1: HelloWorld function

```
void HelloWorld(int count)
{
    if (count<1) return;
    cout<<"Hello World"<<endl;
    HelloWorld(n-1);

}
```

Code 1-1 : HelloWorld function

What will be the output of this function ?
To understand how recursion works, recall that when function call happens, a stack frame is prepared, this stack frame mainly contains:
- The return address—that is where to go after the function finishes.
- Parameters.
- Local variables.
- Other information related to machine state.

The stack frame is pushed onto the call stack. This is known as ***winding***. After the function finishes, the stack frame is popped out of the call stack. This is known as ***unwinding***.

You can trace the recursive call of your function and how it’s pushed / popped from the stack using debugging mode as it appears in *Figure1.1* below.

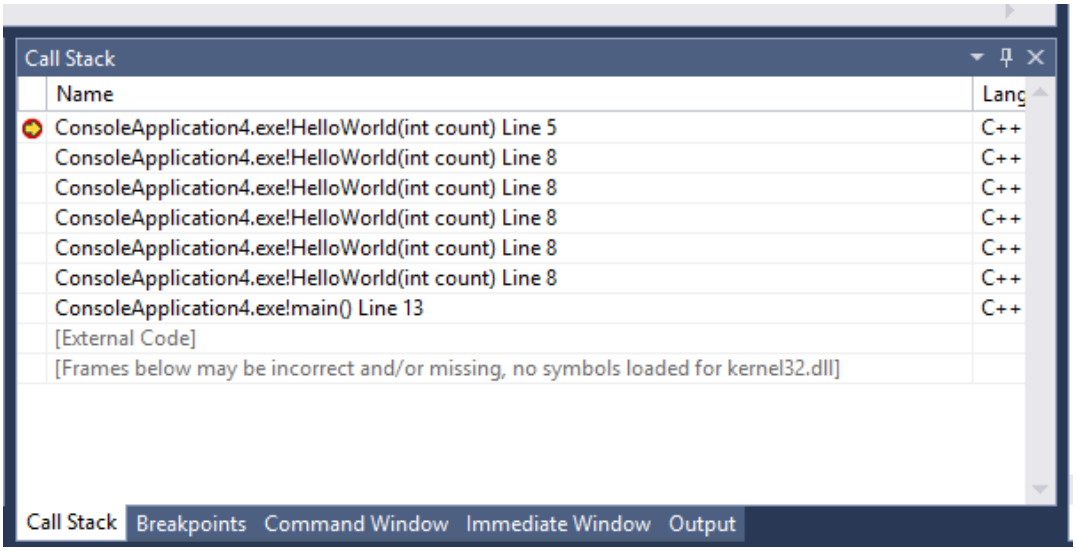

Figure1-1 call stack

How to think recursively?

Recursion can be applied to pretty much any problem, but there are certain scenarios for which you'll find it's particularly helpful. To do so there are some key considerations in designing a recursive algorithm (Reference)

1. **It handles a simple "base case" without using recursion.**

In this example, the base case is "HelloWorld(0)"; if the function is asked to print zero times then it returns without spawning any more "HelloWorld" function calls.

2. **Be sure to avoid cycles**

In many recursive programs, you can avoid cycles or infinite calls by having each function call be for a problem that is somehow smaller or simpler than the original problem, so making sure that we're dealing with progressively smaller or simpler problems till we reach the base case.

3. **Each call of the function represents a complete handling of the given task.**

After you successfully break down the problem to smaller ones make sure that solving the smaller ones finds a solution for bigger ones, we usually end up with small equation that works for any sub problem as well as bigger ones.

Example2: Factorial function

```
int Factorial(int n)
{
   if (n==0)  return 1;
   else return n*Factorial(n-1);

}
```

Code 1-2 : Factorial function

What are the three components for factorial function ?

See Examples➔Rec_BST_Lab.sln➔ 1-Recursion

Part II –Binary Search Tree

Binary Search tree

Binary search tree (BST) is a data structure used for both searching and sorting. Any node contains at most two children and for each node all the values in the left sub-tree are smaller than the root while all the values in the left sub-tree are larger than the root.

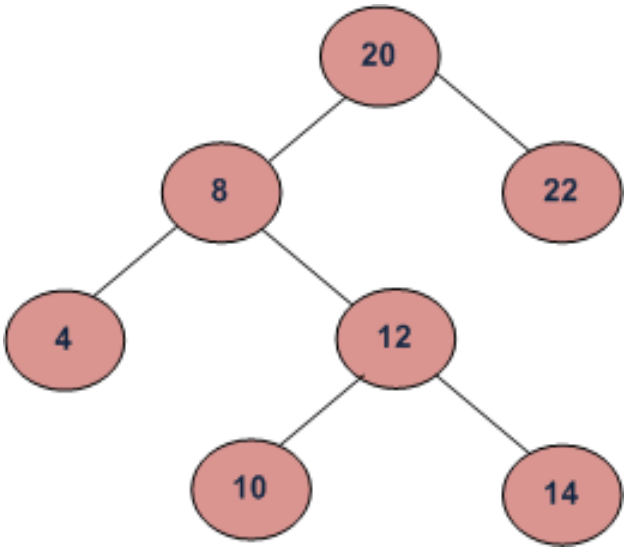

Figure 2-1 Binary search tree

Binary Node Class

The Node class of the binary tree can be defined as follows:

```
class Node
{private:
      int data;
      Node *left;
      Node *right;
public:
      Node(int val);
      void setdata(int d);
      int getdata();
      void setright(Node* p);
      Node*& getright();
      //returns a pointer by reference
      //Reason for that is explained in file BSTree.cpp
      // as a  note written inside the body of function
      //    BSTree::rec_insertBST
      void setleft(Node* p);
      Node*& getleft();
};
```

Code 2-1 : Binary Node class

Binary Search Tree ADT Class

The binary search tree can be defined as follows:

```
class BSTree
{
        Node * root;
        // some utility functions
      public:
        // some public functions
};
```

Code 2-2 : Binary Search Tree class

The tree is a **recursive** data structure by nature which means any node in the tree can be treated as another tree so most functions that deal with trees are written in a recursive manner.

Note:

Each public function in BST will need a utility (private) recursive function that takes a passed sub-root as a parameter. The utility recursive function will operate on the subtree headed at the sub-root passed to the function.

See Examples➔DS_Lab5.sln➔BinarySearchTree

Read the functions carefully and in each function notice the following:

- In each public function, **is it constant function or not? Why?**
- In each utility function, **does it need the "subRoot" to be passed by value or reference? Why?**

Here are the main operations of BST.

BST insertion

Read the code of `insertBST` public function and `rec_insertBST` recursive function.

What will the tree look like if the input data inserted is sorted? How can we solve this problem?

What is the complexity for this insertion algorithm? And What is the searching complexity for BST?

Tree Traversal

Traversing a tree involves iterating (looping) over all nodes in some manner. Three famous traversal algorithms are known for the trees.

1. **In order** tree traversal
2. **Post order** tree traversal
3. **Preorder** tree traversal

Read the code of `inorder_traverse`, `preorder_traverse` and `postorder_traverse` public function and their utility recursive function.

Destroying the Tree

Trees are made of pointers so you should free the memory used by the tree safely before exiting your program. Read **destroy_tree** function and its recursive utility function.

Note: we called destrpy_tree in the BSTree destructor.

Searching

Searching for a node is similar to inserting a node. We start from root, and then go left or right until we find (or not find the node).

Practice Exercises

Part 1: Recursion

1- Write a recursive function ***RecursiveSum*** that computes the sum of all numbers from 1 to n, where n is given as input from user in the main which then calls the function and print the value returned.

2- Write a recursive function ***RecursiveFibonacci*** that calculates and returns the fibonnacci sequence value for a given integer number. Test your function from main by taking an integer input from the user, call ***RecursiveFibonacci*** function and print the results returned.

e.g. fibonacci(0) = 0 , fibonnaci (1) = 1
fibonacci sequence: 0 1 1 2 3 5 8 13 21 …
fibonacci(3) should return 2, fibonacci(4) should return 3, fibonacci(5) should return 5, fibonacci(6) should return 8 …..

3- Write a recursive function ***RecursivePrintArr*** (use no while loops or for loops) that prints all the elements of an array of integers, one per line. The parameters to the function should be int arr[ ], and int size.

4- Write a **member** recursive function ***RecursivePrintLList*** that prints the elements of class linked list.

5- Write a recursive function ***ZeroCount*** to Count the number of zeros in an array of integers recursively.

6- Repeat the previous problem but for linked list.

7- Write a recursive function ***GetMin*** to Find the minimum element in an array of integers.

8- The greatest common devisor of any two numbers (n, m) where m is greater than n, can be calculated according to the Euclidian equation GCD(n,m) = GCD(m, n mod m) and GCD(n, 0)= n .
You are to implement a recursive function to find the GCD of any two numbers taken from the user, call it from the main and print the result in the main.

[Hint the user can enter the two numbers in any order].

9- Write a recursive function to determine how many digits a positive integer has, the integer is taken from the user, call it from the main and print the result in the main. [Hint you can find how many digits by dividing by 10 for example 145/10=14.5 , 14/10=1.4 , 1/10= 0.1]. [Hint know the difference between integer division and float division]

Part 2: Trees

A. Binary Search Trees

Using the function prototypes in BSTree.h:

A.1. Implement and call member function ***find_max*** that returns a pointer to the node that contains the maximum element in a binary search tree.

A.2. Implement and call member function ***find_min*** that returns a pointer to the node that contains the maximum element in a binary search tree.

A.3. Implement and call member function ***max_depth*** that returns an integer representing the maximum depth of a binary search tree.

A.4. Write a member function **search** that searches for a specific value in BSTree.

A.5. Write a member function that returns number of nodes less than a specific value in BST tree

A.6. Write a function ***PrintSmallerEq***(int x) that outputs, in **ascending** order of keys, all elements in a BST whose keys are smaller than or equal to x and returns true. if there were no smaller keys than x then return false.

B. General Binary Trees

Take Node.h and Node.cpp from the examples and make a new class called BinaryTree that contains the following member functions.

[Simple Traversal Problems]:

B.1. Write member function that double the value of each node in the tree.

B.2. Write member function **printLeaves** that prints the leaves of the tree.

B.3. Write member function **printParents** that prints all parent nodes.

B.4. Write member function **CountGreater** that counts number of nodes that has value greater than the passed value.

C. Extra Problems

[Insertion Problems]:

C.1. Write member function **insert_to_fill** that takes a value and inserts a new node with this value in the tree. It traverses the tree in a depth-first pre-order traversal. During this traversal, the first met node that has exactly one child (not 0 or 2) will be the parent of the new node and the new node will be inserted in place of its empty child. If the tree is empty, the function inserts at root. If all the nodes in the tree has either both children or no children, the function shouldn't insert any nodes and returns false, otherwise, returns true.

C.2. Write member function **insertAt** that takes 3 parameters:

a. An integer value, **parentVal**.
b. A character, **loc**, that can have one of these values: **'L'** for left and **'R'** for right.
c. An integer value, **newVal**, which is the value of the new node

The function first searches for the parent node (the node that has the value of **parentVal**) and inserts the new node to its left if the 'loc' is 'L' and to its right if the 'loc' is 'R' and returns true. If the parentVal is not found, the function returns false.

Consider these special cases:

- If the tree is empty, add the new value at the root no matter what the value of ParentVal or 'loc'.
- If the 'loc' child of parent node is not empty (contains another sub-tree), the new node will be inserted at the 'loc' child and the previous existing sub-tree will be in the 'loc' of the new node. See the examples below.
- **I**f more than one node contains parentVal, use only the first found node.

Hint: use a helper function **rec_search** that takes a value and searches for it in the tree and returns a **pointer** of the node that contains this value. Use **pre-order** traversal.

For example, in the following tree:

```
    10
2       5
      4
```

After Tree1.insertAt(5, 'R' , 30) :

```
    10
2       5
      4   30
```

After Tree1.insertAt(10, 'R' , 50) :
(Note that the previous right subtree is inserted to the right (the 'loc') of the new node)

```
    10
2       50
            5
```

4 30

After Tree1.insertAt(10, 'L' , 80) :

(Note that the previous left subtree is inserted to the left (the 'loc') of the new node)

10

80 50

2 5

4 30

[Deletion Problems]:

C.3. Write member function ***Autumn*** that deletes all current leaf nodes in a binary search tree

for example if the tree was like

10

8 12

5 9 11 15

then after calling the function it will be like

10

8 12

C.4. Write member function that deletes all leaf nodes that has even values.

C.5. Write member function that makes a pre-order traversal in the tree and deletes the first met node that has **exactly one** child (not 0 or 2) and makes its child subtree in the place of it in the tree.

[Other Problems]:

C.6. Write member function **IsIdentical** that takes another tree and returns true if the current tree and the passed tree are identical (same in structure and in the values of each node), otherwise, returns false.

C.7. Write member function **IsMirror** that takes another tree and returns true if the current tree is the mirror of the passed tree. A tree is the mirror of another tree if each node in the original tree is switched locations with its sibling (the right child of each node in the first tree is the left child of it in the second tree and vice versa). For example, the following 2 trees are mirrors:

5 6 7 9 10 8 3 2	5 7 6 8 10 9 2 3